



## Heap-construction programs

Edelkamp, Stefan; Elmasry, Amr; Katajainen, Jyrki

*Publication date:*  
2016

*Document version*  
Publisher's PDF, also known as Version of record

*Document license:*  
[Unspecified](#)

*Citation for published version (APA):*  
Edelkamp, S., Elmasry, A., & Katajainen, J. (2016). *Heap-construction programs*. Department of Computer Science, University of Copenhagen. CPH STL Report Vol. 2016 No. 2

# Heap-Construction Programs

Stefan Edelkamp<sup>1</sup>, Amr Elmasry<sup>2</sup>, and Jyrki Katajainen<sup>3</sup>

<sup>1</sup> *Institute for Artificial Intelligence, University Bremen, Am  
Fallturm 1, 28359 Bremen, Germany; [edelkamp@tzi.de](mailto:edelkamp@tzi.de)*

<sup>2</sup> *Department of Computer Engineering and Systems, Alexandria  
University, Alexandria 21544, Egypt; [elmasry@alexu.edu.eg](mailto:elmasry@alexu.edu.eg)*

<sup>3</sup> *Department of Computer Science, University of Copenhagen,  
Universitetsparken 5, 2100 Copenhagen East, Denmark;  
[jyrki@di.ku.dk](mailto:jyrki@di.ku.dk)*

**Abstract.** This report together with an accompanying `tar` ball contains the source code of the programs described and benchmarked in the paper “Heap Construction—50 Years Later”. The programs in this package describe the state of the art in heap construction in 2016.

**Keywords.** Data structures, binary heaps, heap construction, algorithm engineering, element comparisons, element moves, cache misses, branch mispredictions

### **Copyright notice**

Copyright © 2000–2016 by The authors and Performance Engineering Laboratory (University of Copenhagen)

The programs included in the CPH STL are placed in the public domain. The files may be freely copied and distributed, provided that no changes whatsoever are made. Changes are permissible only if the modified files are given new names, different from the names of existing files in the CPH STL, and only if the modified files are clearly identified as not being part of the library. Usage of the source code in derived works is otherwise unrestricted.

The authors have tried to produce correct and useful programs, but no warranty of any kind should be assumed.

### **Release date**

2016-11-04

## Programs

In the paper “Heap Construction—50 Years Later”, the heap-construction programs considered were characterized as follows:

**stl:** The `make_heap` function that came with our `g++` compiler. On closer inspection, it was seen to rely on the bottom-up `sift_down` policy [5, Exercise 5.2.3–18] (see also [8]). Two of the underlying subroutines passed elements by value, so this resulted in some unnecessary element moves.

**F:** Floyd’s Algol program [3] (**F**) converted into C++. In `sift_down`, this program employed the hole technique, so element swaps were not used.

**basic GM:** Our implementation of the algorithm of Gonnet and Munro [4] (**GM**) using a tournament tree. For an input of size  $N$ , the program used a tournament tree requiring  $2N - 1$  extra space for indices and a temporary output area requiring  $N$  extra space for elements.

**GM:** A space-efficient implementation of **GM**. This program could be configured to operate *in-place* ( $O(1)$  extra space) or *in-situ* ( $O(\lg N)$  extra space). Both versions used  $O(1)$  extra space for elements. The program accepted a tuning parameter  $\gamma$  and set the size of the bottom trees to the closest power of two larger than, or equal to,  $\gamma \lg N / \lg \lg N$ . By default,  $\gamma = 32$ . The in-place variant used a packed array that could store a sequence of integers of equal length compactly in memory. The in-situ variant stored the offsets in an integer array.

**MR:** A space-efficient implementation of the algorithm of McDiarmid and Reed [6] (**MR**). The program accepted a tuning parameter  $\mu$  and set the size of the bottom trees to the closest power of two larger than, or equal to,  $\mu \lg N$ . By default,  $\mu = 16$ . As in the previous program, both cache and branch optimizations were applied. All the elements on the `sift_down` path were moved cyclically first after the final position of the new element was known as proposed in [7]. When making a bottom tree into a heap, the indices of the element array from the interval  $[0..N)$  and those of the packed array from the interval  $[0..S)$  were updated in tandem, which doubled the instruction count for index operations. Also this program could be configured to operate in-place (use a packed array of bit pairs) or in-situ (use an array of bytes).

The following optimization options were considered for Floyd’s program:

**opt<sub>1</sub>** [2]: We made sure that `sift_down` was always called with an odd  $N$ . This way, inside the inner loop, one easy-to-predict branch could be removed.

**opt<sub>2</sub>** [2]: We interpreted the result of an element comparison as an integer and used this value in normal index arithmetic. This way, inside the inner loop, the hard-to-predict branch in “**if** (condition)  $j \leftarrow j + 1$ ” could be replaced with an assignment “ $j \leftarrow j + (\text{condition})$ ”.

**opt<sub>3</sub>** [2]: We did not make any element moves when the element at the root stayed in its original location.

- opt<sub>4</sub>** [1]: We visited the nodes in reverse depth-first order instead of reverse breadth-first order.
- opt<sub>5</sub>** [2]: We made the construction in a single loop by fusioning the two loops in `make_heap` and `sift_down`. Inside this loop, conditional moves were used, but two of the element moves were still made conditionally so that the number of element moves would not increase to  $5N$ . The outcome of these two conditional branches was predicted reasonably well so it was not worth avoiding these branches.

### Portability

In the experiments reported in the paper ‘Heap Construction—50 Years Later’, in each experiment a C array was used to store the input elements. As in our pseudo-code, to access the element with rank `i` in an array `A` one would normally write `A[i]`. However, the parameters given for `std::make_heap` are iterators and for them one has to write `*(A + i)` instead of `A[i]`. For arrays both expressions are valid and the semantics is the same. In this revised version all the programs have been made portable so that they can be used in the same way as C++ standard-library function `make_heap`. However, be aware that in some cases a portable version can be a bit slower than a version using C arrays directly.

### References

- [1] J. Bojesen, J. Katajainen, and M. Spork, Performance engineering case study: Heap construction, *ACM J. Exp. Algorithmics* **5** (2000), Article 15.
- [2] A. Elmasry and J. Katajainen, Lean programs, branch mispredictions, and sorting, *FUN 2012, LNCS* **7288**, Springer, Heidelberg (2012), 119–130.
- [3] R. W. Floyd, Algorithm 245: Treesort 3, *Commun. ACM* **7**, 12 (1964), 701.
- [4] G. H. Gonnet and J. I. Munro, Heaps on heaps, *SIAM J. Comput.* **15**, 4 (1986), 964–971.
- [5] D. E. Knuth, *Sorting and Searching, The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman, Reading (1998).
- [6] C. J. H. McDiarmid and B. A. Reed, Building heaps fast, *J. Algorithms* **10**, 3 (1989), 352–365.
- [7] I. Wegener, The worst case complexity of McDiarmid and Reed’s variant of Bottom-Up Heapsort is less than  $n \log n + 1.1n$ , *Inform. and Comput.* **97**, 1 (1992), 86–96.
- [8] I. Wegener, Bottom-Up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if  $n$  is not very small), *Theoret. Comput. Sci.* **118**, 1 (1993), 81–98.

## Contents

File	Description	Page
<a href="#">stl.h++</a>	The <a href="#">make_heap</a> function that came with our g++ compiler	6
<a href="#">f.h++</a>	A translation of Floyd's Algol program ( <b>F</b> ) [3] into C++	7
<a href="#">opt1.h++</a>	A modification of <b>F</b> employing <b>opt<sub>1</sub></b> [2]	8
<a href="#">opt2.h++</a>	A modification of <b>F</b> employing <b>opt<sub>1</sub></b> and <b>opt<sub>2</sub></b> [2]	10
<a href="#">opt3.h++</a>	A modification of <b>F</b> employing <b>opt<sub>1</sub></b> , <b>opt<sub>2</sub></b> , and <b>opt<sub>3</sub></b> [2]	11
<a href="#">opt4.h++</a>	A modification of <b>F</b> employing <b>opt<sub>1</sub></b> , <b>opt<sub>2</sub></b> , <b>opt<sub>3</sub></b> , and <b>opt<sub>4</sub></b> [1]	13
<a href="#">opt5.h++</a>	A branch-optimized version of <b>F</b> taken from [2]	15
<a href="#">basic_gm.h++</a>	Our implementation of the algorithm of Gonnet and Munro ( <b>GM</b> ) [4] using a tournament tree	17
<a href="#">gm.h++</a>	Our implementation of <b>GM</b> that can be configured to operate in-place or in-situ	23
<a href="#">tuned_gm.h++</a>	A tuned version of <b>GM</b> that operates in-situ, but uses a logarithmic amount of additional space for elements too	30
<a href="#">mr.h++</a>	Our implementation of the algorithm of McDiarmid and Reed ( <b>MR</b> ) [6] that can be configured to operate in-place or in-situ	37
<a href="#">packed_array.h++</a>	A data structure that can store a sequence of integers of equal length compactly in memory	41
<a href="#">driver.c++</a>	The driver used for measuring the CPU time, the number of element comparisons, and the number of element moves	44
<a href="#">test-driver.c++</a>	The driver used for unit testing	50
<a href="#">makefile</a>	This <a href="#">makefile</a> can be used to redo some of the experiments described in the paper; the scripts to derive the results for the number of instructions, cache misses, and branch mispredictions are <i>not</i> included in this catalogue	52

## Heap-construction programs

*stl.h++*

```

1 #include <iterator> // std::iterator_traits
2 #include <utility> // std::move
3
4 namespace stl {
5
6     template <typename iterator, typename index, typename T, typename
7         comparator>
8     void push_heap(iterator a, index hole, index top, T value,
9         comparator less) {
10         index parent = (hole - 1) / 2;
11         while (hole > top and less(*(a + parent), value)) {
12             *(a + hole) = std::move(*(a + parent));
13             hole = parent;
14             parent = (hole - 1) / 2;
15         }
16         *(a + hole) = std::move(value);
17     }
18
19     template <typename iterator, typename index, typename T, typename
20         comparator>
21     void adjust_heap(iterator a, index hole, index n, T value,
22         comparator less) {
23         index const top = hole;
24         index second = hole;
25         while (second < (n - 1) / 2) {
26             second = 2 * (second + 1);
27             if (less(*(a + second), *(a + (second - 1)))) {
28                 second--;
29             }
30             *(a + hole) = std::move(*(a + second));
31             hole = second;
32         }
33         if ((n bitand 1) == 0 and second == (n - 2) / 2) {
34             second = 2 * (second + 1);
35             *(a + hole) = std::move(*(a + (second - 1)));
36             hole = second - 1;
37         }
38         stl::push_heap(a, hole, top, std::move(value), less);
39     }
40
41     template <typename iterator, typename comparator>
42     void make_heap(iterator a, iterator past, comparator less) {
43         using element = typename std::iterator_traits<iterator>::
44             value_type;
45         using index = typename std::iterator_traits<iterator>::
46             difference_type;
47         if (past - a < 2) {
48             return;
49         }
50     }
51 }

```

```

43     }
44     index const n = past - a;
45     index parent = (n - 2) / 2;
46     while (true) {
47         element value = std::move(*(a + parent));
48         stl::adjust_heap(a, parent, n, std::move(value), less);
49         if (parent == 0) {
50             return;
51         }
52         -- parent;
53     }
54 }
55 }

```

*f.h++*

```

1 // Floyd's original Algol program translated into C++
2
3 #include <cstddef> // std::size_t
4 #include <iterator> // std::iterator_traits
5 #include <utility> // std::move
6
7 namespace f {
8
9     constexpr std::size_t root() {
10         return 0;
11     }
12
13     template <typename index>
14     index parent(index i) {
15         return (i - 1) / 2;
16     }
17
18     template <typename index>
19     index left_child(index i) {
20         return (i << 1) + 1;
21     }
22
23     template <typename iterator, typename index, typename comparator>
24     void sift_down(iterator a, index i, index n, comparator less) {
25         using element = typename std::iterator_traits<iterator>::
26             value_type;
27         element x = std::move(*(a + i));
28     loop:
29         index j = left_child(i);
30         if (j < n) {
31             if (j < (n - 1)) {
32                 if (less(*(a + j), *(a + (j + 1)))) {
33                     j = j + 1;
34                 }
35             }
36             if (less(x, *(a + j))) {

```



```

36         *(a + i) = std::move(*(a + j));
37         i = j;
38         goto loop;
39     }
40 }
41 *(a + i) = std::move(x);
42 }
43
44 template <typename iterator, typename comparator>
45 void make_heap(iterator first, iterator past, comparator less) {
46     using index = std::size_t;
47     index const n = past - first;
48     if (n < 2) {
49         return;
50     }
51     index i = parent(n - 1);
52     while (true) {
53         sift_down(first, i, n, less);
54         if (i == root()) {
55             break;
56         }
57         -- i;
58     }
59 }
60 }

```

*opt1.h++*

```

1 // Floyd's heap-construction program
2
3 // Optimization: one easy-to-predict branch removed
4
5 #include <algorithm> // std::make_heap
6 #include <cstdint> // std::size_t
7 #include <iterator> // std::iterator_traits
8 #include <utility> // std::move
9
10 namespace opt1 {
11
12     constexpr std::size_t root() {
13         return 0;
14     }
15
16     template <typename index>
17     index parent(index i) {
18         return (i - 1) / 2;
19     }
20
21     template <typename index>
22     index left_child(index i) {
23         return (i << 1) + 1;
24     }

```

```

25
26 template <typename iterator, typename index, typename comparator>
27 void sift_up(iterator a, index j, comparator less) {
28     using element = typename std::iterator_traits<iterator>::
        value_type;
29     element in = std::move(*(a + j));
30     while (j > root()) {
31         index i = parent(j);
32         if (less(*(a + i), in)) {
33             *(a + j) = std::move(*(a + i));
34             j = i;
35         }
36         else {
37             break;
38         }
39     }
40     *(a + j) = std::move(in);
41 }
42
43 template <typename iterator, typename index, typename comparator>
44 void sift_down(iterator a, index i, index n, comparator less) {
45     using element = typename std::iterator_traits<iterator>::
        value_type;
46     element in = std::move(*(a + i));
47 loop:
48     index j = left_child(i);
49     if (j < n) {
50         if (less(*(a + j), *(a + (j + 1)))) {
51             j = j + 1;
52         }
53         if (less(in, *(a + j))) {
54             *(a + i) = std::move(*(a + j));
55             i = j;
56             goto loop;
57         }
58     }
59     *(a + i) = std::move(in);
60 }
61
62 template <typename iterator, typename comparator>
63 void make_heap(iterator first, iterator past, comparator less) {
64     using index_t = std::size_t;
65     index const n = past - first;
66     if (n < 3) {
67         std::make_heap(first, past, less);
68         return;
69     }
70     index const m = (n bitand 1) ? n : n - 1;
71     index i = parent(m - 1);
72     while (true) {
73         sift_down(first, i, m, less);
74         if (i == root()) {

```

```

75         break;
76     }
77     -- i;
78 }
79 sift_up(first, n - 1, less);
80 }
81 }

```

*opt2.h++*

```

1 // Floyd's heap-construction program
2
3 // Optimizations: 1) one easy-to-predict branch removed; 2) one
4 // hard-to-predict branch removed
5
6 #include <cstddef> // std::size_t
7 #include <iterator> // std::iterator_traits
8 #include <utility> // std::move
9
10 namespace opt2 {
11
12     constexpr std::size_t root() {
13         return 0;
14     }
15
16     template <typename index>
17     index parent(index i) {
18         return (i - 1) / 2;
19     }
20
21     template <typename index>
22     index left_child(index i) {
23         return (i << 1) + 1;
24     }
25
26     template <typename iterator, typename index, typename comparator>
27     void sift_up(iterator a, index j, comparator less) {
28         using element = typename std::iterator_traits<iterator>::
29             value_type;
30         element in = std::move(*(a + j));
31         while (j > root()) {
32             index i = parent(j);
33             if (less(*(a + i), in)) {
34                 *(a + j) = std::move(*(a + i));
35                 j = i;
36             }
37             else {
38                 break;
39             }
40         }
41         *(a + j) = std::move(in);
42     }
43 }

```

```

42
43 template <typename iterator, typename index, typename comparator>
44 void sift_down(iterator a, index i, index n, comparator less) {
45     using element = typename std::iterator_traits<iterator>::
         value_type;
46     element in = std::move(*(a + i));
47 loop:
48     index j = left_child(i);
49     if (j < n) {
50         j = j + less(*(a + j), *(a + (j + 1)));
51         if (less(in, *(a + j))) {
52             *(a + i) = std::move(*(a + j));
53             i = j;
54             goto loop;
55         }
56     }
57     *(a + i) = std::move(in);
58 }
59
60 template <typename iterator, typename comparator>
61 void make_heap(iterator first, iterator past, comparator less) {
62     using index = std::size_t;
63     index const n = past - first;
64     if (n < 3) {
65         std::make_heap(first, past, less);
66         return;
67     }
68     index const m = (n & 1) ? n : n - 1;
69     index i = parent(m - 1);
70     while (true) {
71         sift_down(first, i, m, less);
72         if (i == root()) {
73             break;
74         }
75         -- i;
76     }
77     sift_up(first, n - 1, less);
78 }
79 }

```

### *opt3.h++*

```

1 // Floyd's heap-construction program
2
3 // Optimizations: 1) one easy-to-predict branch removed; 2) one
4 // hard-to-predict branch removed; 3) no element moves done in
5 // sift_down if the new element can stay at the root
6
7 #include <algorithm> // std::make_heap
8 #include <cstdint> // std::size_t
9 #include <iterator> // std::iterator_traits
10 #include <utility> // std::move

```

```

11
12 namespace opt3 {
13
14     constexpr std::size_t root() {
15         return 0;
16     }
17
18     template <typename index>
19     index parent(index i) {
20         return (i - 1) / 2;
21     }
22
23     template <typename index>
24     index left_child(index i) {
25         return (i << 1) + 1;
26     }
27
28     template <typename iterator, typename index, typename comparator>
29     void sift_up(iterator a, index j, comparator less) {
30         using element = typename std::iterator_traits<iterator>::
            value_type;
31         element in = std::move(*(a + j));
32         while (j > root()) {
33             index i = parent(j);
34             if (less(*(a + i), in)) {
35                 *(a + j) = std::move(*(a + i));
36                 j = i;
37             }
38             else {
39                 break;
40             }
41         }
42         *(a + j) = std::move(in);
43     }
44
45     template <typename iterator, typename index, typename comparator>
46     void sift_down(iterator a, index i, index n, comparator less) {
47         using element = typename std::iterator_traits<iterator>::
            value_type;
48         index j = left_child(i);
49         j = j + less(*(a + j), *(a + (j + 1)));
50         if (not less(*(a + i), *(a + j))) {
51             return;
52         }
53         element in = std::move(*(a + i));
54         *(a + i) = std::move(*(a + j));
55     loop:
56         i = j;
57         j = left_child(i);
58         if (j < n) {
59             j = j + less(*(a + j), *(a + (j + 1)));
60             if (less(in, *(a + j))) {

```

```

61         *(a + i) = std::move(*(a + j));
62         goto loop;
63     }
64 }
65 *(a + i) = std::move(in);
66 }
67
68 template <typename iterator, typename comparator>
69 void make_heap(iterator first, iterator past, comparator less) {
70     using index = std::size_t;
71     index const n = past - first;
72     if (n < 3) {
73         std::make_heap(first, past, less);
74         return;
75     }
76     index const m = (n & 1) ? n : n - 1;
77     index i = parent(m - 1);
78     while (true) {
79         sift_down(first, i, m, less);
80         if (i == root()) {
81             break;
82         }
83         -- i;
84     }
85     sift_up(first, n - 1, less);
86 }
87 }

```

#### *opt4.h++*

```

1 // Floyd's heap-construction program
2
3 // Optimizations: 1) one easy-to-predict branch removed; 2) one
4 // hard-to-predict branch removed; 3) no element moves done in
5 // sift_down if the new element can stay at the root; 4) nodes are
6 // visited in depth-first order to improve the cache behaviour
7
8 #include <algorithm> // std::make_heap
9 #include <cmath> // ilogb
10 #include <cstddef> // std::size_t
11 #include <iterator> // defines std::iterator_traits
12 #include <utility> // std::move std::swap
13
14 namespace opt4 {
15
16     constexpr std::size_t root() {
17         return 0;
18     }
19
20     template <typename index>
21     index parent(index i) {
22         return (i - 1) / 2;
23     }
24 }

```

```

23     }
24
25     template <typename index>
26     index left_child(index i) {
27         return (i << 1) + 1;
28     }
29
30     template <typename iterator, typename index, typename comparator>
31     void sift_up(iterator a, index j, comparator less) {
32         using element = typename std::iterator_traits<iterator>::
            value_type;
33         element in = std::move(*(a + j));
34         while (j > root()) {
35             index i = parent(j);
36             if (less(*(a + i), in)) {
37                 *(a + j) = std::move(*(a + i));
38                 j = i;
39             }
40             else {
41                 break;
42             }
43         }
44         *(a + j) = std::move(in);
45     }
46
47     template <typename iterator, typename index, typename comparator>
48     void sift_down_1_3(iterator a, index i, index n, comparator less) {
49         index j = left_child(i);
50         if (j ≥ n) {
51             return;
52         }
53         j = j + less(*(a + j), *(a + (j + 1)));
54         if (not less(*(a + i), *(a + j))) {
55             return;
56         }
57         std::swap(*(a + i), *(a + j));
58     }
59
60     template <typename iterator, typename index, typename comparator>
61     void sift_down(iterator a, index i, index n, comparator less) {
62         using element = typename std::iterator_traits<iterator>::
            value_type;
63         index j = left_child(i);
64         j = j + less(*(a + j), *(a + (j + 1)));
65         if (not less(*(a + i), *(a + j))) {
66             return;
67         }
68         element in = std::move(*(a + i));
69     loop:
70         *(a + i) = std::move(*(a + j));
71         i = j;
72         j = left_child(j);

```

```

73     if (j < n) {
74         j = j + less(*(a + j), *(a + (j + 1)));
75         if (less(in, *(a + j))) {
76             goto loop;
77         }
78     }
79     *(a + i) = std::move(in);
80 }
81
82 template <typename iterator, typename comparator>
83 void make_heap(iterator first, iterator past, comparator less) {
84     using index = std::size_t;
85     index const n = past - first;
86     if (n < 5) {
87         std::make_heap(first, past, less);
88         return;
89     }
90     index const m = (n & 1) ? n : n - 1;
91     index j = (1 << ilogb(m)) - 2;
92     index const i = parent(j + 1);
93     while (j ≥ i) {
94         sift_down_1_3(first, j, m, less);
95         index z = j;
96         while ((z & 1) == 1) {
97             z = parent(z);
98             sift_down(first, z, m, less);
99         }
100         -- j;
101     }
102     sift_up(first, n - 1, less);
103 }
104 }

```

### *opt5.h++*

```

1  // A program that constructs a binary heap in a single loop
2
3  // Optimizations: 1) branches avoided in general; 2) a compiler
4  // should use conditional moves, not conditional branches;
5  // 3) construction is done in a single loop to avoid the branch
6  // mispredictions caused by the inner loop when stepping out of it
7
8  #include <algorithm> // std::make_heap
9  #include <cstddef> // std::size_t
10 #include <iterator> // std::iterator_traits
11 #include <utility> // std::move
12
13 namespace opt5 {
14
15     constexpr std::size_t root() {
16         return 0;
17     }

```



```

18
19 template <typename index>
20 index parent(index i) {
21     return (i - 1) / 2;
22 }
23
24 template <typename index>
25 index left_child(index i) {
26     return (i << 1) + 1;
27 }
28
29 template <typename iterator, typename index, typename comparator>
30 void sift_up(iterator a, index j, comparator less) {
31     using element = typename std::iterator_traits<iterator>::
32         value_type;
33     element in = std::move(*(a + j));
34     while (j > root()) {
35         index i = parent(j);
36         if (less(*(a + i), in)) {
37             *(a + j) = std::move(*(a + i));
38             j = i;
39         }
40         else {
41             break;
42         }
43     }
44     *(a + j) = std::move(in);
45 }
46
47 template <typename iterator, typename comparator>
48 void make_heap(iterator first, iterator past, comparator less) {
49     using index = std::size_t;
50     using element = typename std::iterator_traits<iterator>::
51         value_type;
52     index const n = past - first;
53     if (n < 3) {
54         std::make_heap(first, past, less);
55         return;
56     }
57     index const m = (n & 1) ? n : n - 1;
58     index i = parent(m - 1);
59     index j = i;
60     index hole = j;
61     element in = std::move(*(first + j));
62     while (true) {
63         hole = (i == j) ? j : hole;
64         if (i == j) {
65             in = std::move(*(first + j));
66         }
67         j = left_child(j);
68         j += less(*(first + j), *(first + (j + 1)));
69         *(first + hole) = std::move(*(first + j));

```

```

68     hole = less(in, *(first + j)) ? j : hole;
69     bool done = (left_child(j) ≥ m);
70     if (done) {
71         *(first + hole) = std::move(in);
72     }
73     if (done and i == root()) {
74         break;
75     }
76     i = (done) ? i - 1 : i;
77     j = (done) ? i : j;
78 }
79 sift_up(first, n - 1, less);
80 }
81 }

```

### *basic\_gm.h++*

```

1 // Gonnet & Munro
2
3 #include <algorithm> // std::copy std::make_heap
4 #include <cstdint> // std::size_t
5 #include <functional> // std::less
6 #include <iterator> // std::iterator_traits
7
8 namespace basic_gm {
9
10     constexpr std::size_t root() {
11         return 0;
12     }
13
14     template <typename index>
15     index parent(index i) {
16         return (i - 1) / 2;
17     }
18
19     template <typename index>
20     index left_child(index i) {
21         return 2 * i + 1;
22     }
23
24     template <typename index>
25     index right_child(index i) {
26         return 2 * i + 2;
27     }
28
29     template <typename index>
30     index odd(index i) {
31         return (i bitand 1) == 1;
32     }
33
34     template <typename index>
35     index even(index i) {

```

```

36     return (i & 1) == 0;
37 }
38
39 template <typename index>
40 index sibling(index i) {
41     return i + odd(i) - even(i);
42 }
43
44 template <typename index>
45 bool is_inside(index i, index n) {
46     return i < n;
47 }
48
49 template <typename integer>
50 bool is_power_of_2(integer n) {
51     return __builtin_popcount(n) == 1;
52 }
53
54 template <typename iterator, typename index, typename comparator>
55 void sift_down(iterator a, index i, index n, comparator less) {
56     using element = typename std::iterator_traits<iterator>::
57         value_type;
58     element x = std::move(*(a + i));
59     index j = left_child(i);
60     while (is_inside(j, n)) {
61         if (is_inside(j + 1, n)) {
62             j = j + less(*(a + j), *(a + (j + 1)));
63         }
64         if (not less(x, *(a + j))) {
65             break;
66         }
67         *(a + i) = std::move(*(a + j));
68         i = j;
69         j = left_child(i);
70     }
71     *(a + i) = std::move(x);
72 }
73
74 template <typename iterator, typename index, typename comparator>
75 void make_heap(iterator a, index K, index N, comparator less) {
76     if (not is_inside(left_child(K), N)) {
77         return;
78     }
79     index L = K;
80     while (is_inside(left_child(L), N)) {
81         L = left_child(L);
82     }
83     index R = K;
84     while (is_inside(right_child(R), N)) {
85         R = right_child(R);
86     }
87     if (R < L) {

```

```

87     R = right_child(R);
88 }
89 do {
90     L = parent(L);
91     R = parent(R);
92     index J = R;
93     while (true) {
94         sift_down(a, J, N, less);
95         if (J == L) {
96             break;
97         }
98         J -= 1;
99     }
100 } while (L != R);
101 }
102
103 template <typename input, typename index, typename output>
104 void copy_heap(input a, index k, index h, output o) {
105     index i = k;
106     while (true) {
107         index j = i;
108         while (j ≤ k) {
109             *(o + j) = std::move(*(a + j));
110             ++j;
111         }
112         if (h == 0) {
113             return;
114         }
115         --h;
116         i = left_child(i);
117         k = right_child(k);
118     }
119 }
120
121 template <typename tournament, typename index>
122 void populate_tournament(tournament t, index n, index K, index L,
123     index R, index E) {
124     index i = parent(n);
125     t[i] = E;
126     ++i;
127     index I = R;
128     while (I ≥ L) {
129         t[i] = I;
130         ++i;
131         index Z = I;
132         while (odd(Z) and Z ≠ K) {
133             Z = parent(Z);
134             t[i] = Z;
135             ++i;
136         }
137         --I;
138     }

```

```

138 }
139
140 template <typename tournament, typename index, typename input,
        typename comparator>
141 void run_tournament(tournament t, index n, input a, comparator less
        ) {
142     index i = parent(n - 1);
143     while (true) {
144         index C = t[left_child(i)];
145         index D = t[right_child(i)];
146         t[i] = less(*(a + C), *(a + D)) ? D : C;
147         if (i == root()) {
148             break;
149         }
150         -- i;
151     }
152 }
153
154 template <typename tournament, typename index, typename input,
        typename comparator>
155 void update(tournament t, index j, index h, index E, input a,
        comparator less) {
156     index M = t[j]; // location of the old champion
157     index k = j;
158     do {
159         k = left_child(k);
160         k = k + (t[k] != M);
161         -- h;
162     } while (h != 0);
163     t[k] = E;
164     do {
165         k = parent(k);
166         index C = t[left_child(k)];
167         index D = t[right_child(k)];
168         t[k] = less(*(a + C), *(a + D)) ? D : C;
169     } while (k != j);
170 }
171
172 template <typename tournament, typename index, typename input,
        typename output, typename comparator>
173 index handle_base_case(tournament t, index j, input a, output o,
        index J, comparator less) {
174     index champion = t[j];
175     *(o + J) = std::move(*(a + champion));
176     index k = left_child(j);
177     index l = k + (t[k] == champion);
178     index w = sibling(l);
179     index L = left_child(J);
180     index W = right_child(J);
181     index second = t[l];
182     *(o + L) = std::move(*(a + second));
183     k = left_child(l);

```

```

184     index ll = k + (t[k] == second);
185     index lw = sibling(ll);
186     *(o + left_child(L)) = std::move(*(a + t[ll]));
187     k = left_child(lw);
188     index lwl = k + (t[k] == second);
189     *(o + right_child(L)) = std::move(*(a + t[lwl]));
190     k = left_child(ll);
191     index lll = k + (t[k] == t[ll]);
192     k = left_child(w);
193     index wl = k + (t[k] == champion);
194     index ww = sibling(wl);
195     k = left_child(ww);
196     index wwl = k + (t[k] == champion);
197     index excess = t[wwl];
198     k = left_child(wl);
199     index wll = k + (t[k] == t[wl]);
200     *(o + left_child(W)) = std::move(*(a + t[wll]));
201     index U = t[wl];
202     index V = t[lll];
203     if (less(*(a + U), *(a + V))) {
204         *(o + W) = std::move(*(a + V));
205         *(o + right_child(W)) = std::move(*(a + U));
206     }
207     else {
208         *(o + W) = std::move(*(a + U));
209         *(o + right_child(W)) = std::move(*(a + V));
210     }
211     return excess;
212 }
213
214 template <typename tournament, typename index, typename input,
215           typename output, typename comparator>
216 index convert_tournament(tournament t, index j, index h, input a,
217                          output o, index J, comparator less) {
218     // J: current output location in the heap
219     // j: current root in the tournament tree
220     if (h == 3) {
221         return handle_base_case(t, j, a, o, J, less);
222     }
223     index champion = t[j];
224     *(o + J) = std::move(*(a + champion));
225     index k = left_child(j);
226     k = k + (t[k] == champion);
227     index excess = convert_tournament(t, k, h - 1, a, o, left_child(J), less);
228     k = sibling(k);
229     update(t, k, h - 1, excess, a, less);
230     return convert_tournament(t, k, h - 1, a, o, right_child(J), less);
231 }

```

```

231 template <typename input, typename index, typename tournament,
      typename output, typename comparator>
232 index make_heap(input a, index K, index N, index E, tournament t,
      output o, comparator less) {
233     if (not is_inside(K, N)) {
234         return E;
235     }
236     index h = 0;
237     index L = K;
238     while (is_inside(left_child(L), N)) {
239         ++h;
240         L = left_child(L);
241     }
242     index height = h;
243     index R = K;
244     h = 0;
245     while (is_inside(right_child(R), N)) {
246         ++h;
247         R = right_child(R);
248     }
249     index n = 1 << (height + 1);
250     if (n < 8) {
251         copy_heap(a, K, h, o);
252         make_heap(o, K, N, less);
253         return E;
254     }
255     populate_tournament(t, 2 * n - 1, K, L, R, E);
256     run_tournament(t, 2 * n - 1, a, less);
257     E = convert_tournament(t, root(), height + 1, a, o, K, less);
258     return E;
259 }
260
261 template <typename iterator, typename comparator>
262 void make_heap(iterator a, iterator past, comparator less) {
263     using index = std::size_t;
264     using element = typename std::iterator_traits<iterator>::
      value_type;
265     index const n = past - a;
266     if (n < 2) {
267         return;
268     }
269     index* t = (index*) malloc((((4 * n) / 3) * sizeof(index)));
270     element* o = (element*) malloc(n * sizeof(element)); // no
      element moves
271     index i = n; // invariant: everything moved up to this point
272     if (odd(n)) {
273         *(o + n - 1) = std::move(*(a + n - 1));
274         i = n - 1;
275     }
276     do {
277         index j = sibling(i);
278         i = parent(i);

```

```

279     index excess = i;
280     excess = make_heap(a, j, n, excess, t, o, less);
281     o[i] = std::move(*(a + excess));
282 } while (i ≠ root());
283 i = n - 1;
284 do {
285     i = parent(i);
286     sift_down(o, i, n, less);
287 } while (i ≠ root());
288 for (index i = 0; i ≠ n; ++i) {
289     *(a + i) = std::move(*(o + i));
290 }
291 free(t);
292 free(o);
293 }
294 }

```

*gm.h++*

```

1 // In-situ/In-place Gonnet & Munro
2
3 // #define IN_PLACE
4
5 #include <cstdint> // std::size_t
6 #include <iterator> // std::iterator_traits
7 #include <vector>
8 #include <utility> // std::move
9
10 #ifdef IN_PLACE
11 #include "packed_array.h++" // cphstl::packed_array
12 #endif
13
14 #ifndef FACTOR
15 #define FACTOR 32
16 #endif
17
18 namespace gm {
19
20     constexpr std::size_t root() {
21         return 0;
22     }
23
24     template <typename index>
25     index parent(index i) {
26         return (i - 1) / 2;
27     }
28
29     template <typename index, typename height>
30     index ancestor(index i, height h) {
31         return (i + 1) / (1 << h) - 1;
32     }
33

```



```

34  template <typename index>
35  index left_child(index i) {
36      return 2 * i + 1;
37  }
38
39  template <typename index>
40  index right_child(index i) {
41      return 2 * i + 2;
42  }
43
44  template <typename index>
45  index odd(index i) {
46      return (i & 1) == 1;
47  }
48
49  template <typename index>
50  index even(index i) {
51      return (i & 1) == 0;
52  }
53
54  template <typename index>
55  index sibling(index i) {
56      return i + odd(i) - even(i);
57  }
58
59  template <typename index>
60  bool is_inside(index i, index n) {
61      return i < n;
62  }
63
64  template <typename index>
65  bool general_case(index i, index n) {
66      return left_child(left_child(left_child(i))) < parent(n);
67  }
68
69  std::size_t ilogb(std::size_t x) {
70      asm("bsr %0, %0\n"
71          : "=r"(x)
72          : "0"(x)
73          );
74      return x;
75  }
76
77  template <typename offset, typename index>
78  index offset_to_index(offset i, index J) {
79      if (index(i) != index(0)) {
80          index h = ilogb(index(i));
81          return (1 << h) * J + index(i) - 1;
82      }
83      return parent(J);
84  }
85

```

```

86 template <typename iterator, typename index, typename comparator>
87 void sift_down(iterator a, index i, index n, comparator less) {
88     using element = typename std::iterator_traits<iterator>::
        value_type;
89     element x = std::move(*(a + i));
90     index j = left_child(i);
91     while (is_inside(j, n)) {
92         if (is_inside(j + 1, n)) {
93             j = j + less(*(a + j), *(a + (j + 1)));
94         }
95         if (not less(x, *(a + j))) {
96             break;
97         }
98         *(a + i) = std::move(*(a + j));
99         i = j;
100        j = left_child(i);
101    }
102    *(a + i) = std::move(x);
103 }
104
105 template <typename iterator, typename index, typename comparator>
106 void make_heap(iterator a, index K, index N, comparator less) {
107     index L = K;
108     while (is_inside(left_child(L), N)) {
109         L = left_child(L);
110     }
111     index R = K;
112     while (is_inside(right_child(R), N)) {
113         R = right_child(R);
114     }
115     if (R < L) {
116         R = right_child(R);
117     }
118     do {
119         L = parent(L);
120         R = parent(R);
121         index J = R;
122         while (true) {
123             sift_down(a, J, N, less);
124             if (J == L) {
125                 break;
126             }
127             J -= 1;
128         }
129     } while (L != R);
130 }
131
132 template <typename tree, typename offset>
133 void populate_tournament(tree& t, offset n) {
134     offset i = 0;
135     for (offset j = parent(n); is_inside(j, n); ++j) {
136         t[j] = i;

```

```

137         i = i + 1;
138     }
139 }
140
141 template <typename tree, typename offset, typename iterator,
142           typename index, typename comparator>
143 void run_tournament(tree& t, offset n, iterator a, index J,
144                   comparator less) {
145     offset j = parent(n - 1);
146     while (true) {
147         offset k = t[left_child(j)];
148         index L = offset_to_index(k, J);
149         k = t[right_child(j)];
150         index R = offset_to_index(k, J);
151         k = left_child(j) + less(*(a + L), *(a + R));
152         t[j] = t[k];
153         if (j == root<offset>()) {
154             break;
155         }
156         j = j - 1;
157     }
158 }
159
160 template <typename tree, typename offset, typename iterator,
161           typename index, typename comparator>
162 void update(tree& t, offset i, offset n, offset excess, iterator a,
163            index J, comparator less) {
164     offset j = i; // root of the subtree considered
165     auto top = t[j]; // offset of the old top
166     t[j] = excess;
167     while (is_inside(left_child(j), n)) {
168         j = left_child(j) + (t[right_child(j)] == top);
169         t[j] = excess;
170     }
171     while (j != i) {
172         j = parent(j);
173         offset k = t[left_child(j)];
174         index L = offset_to_index(k, J);
175         k = t[right_child(j)];
176         index R = offset_to_index(k, J);
177         k = left_child(j) + less(*(a + L), *(a + R));
178         t[j] = t[k];
179     }
180 }
181
182 template <typename tree, typename offset, typename permutation,
183           typename iterator, typename index, typename comparator>
184 offset handle_base_case(tree& t, offset j, permutation& sigma,
185                       iterator a, index J, comparator less) {
186     auto champion = t[j];
187     sigma[j + 1] = champion;
188     offset k = left_child(j);

```

```

183     offset l = k + (t[k] == champion);
184     offset w = sibling(l);
185     auto finalist = t[l];
186     sigma[l + 1] = finalist;
187     k = left_child(l);
188     offset ll = k + (t[k] == finalist);
189     offset lw = sibling(ll);
190     sigma[ll + 1] = t[ll];
191     k = left_child(lw);
192     offset lwl = k + (t[lw] == t[k]);
193     sigma[lw + 1] = t[lwl];
194     k = left_child(ll);
195     offset lll = k + (t[ll] == t[k]);
196     k = left_child(w);
197     offset wl = k + (t[k] == champion);
198     offset ww = sibling(wl);
199     k = left_child(ww);
200     offset wwl = k + (t[k] == champion);
201     offset excess = t[wwl];
202     k = left_child(wl);
203     auto semifinalist = t[wl];
204     offset wll = k + (t[k] == semifinalist);
205     sigma[wl + 1] = t[wll];
206     index K = offset_to_index(semifinalist, J);
207     index L = offset_to_index(t[lll], J);
208     bool line = less(*(a + K), *(a + L));
209     sigma[w + 1] = (line) ? t[lll] : semifinalist;
210     sigma[ww + 1] = (line) ? semifinalist : t[lll];
211     return excess;
212 }
213
214 template <typename tree, typename offset, typename permutation,
215           typename iterator, typename index, typename comparator>
216 void convert_tournament(tree& t, offset n, permutation& sigma,
217                         iterator a, index J, comparator less) {
218
219     #ifdef IN_PLACE
220         cphstl::packed_array stack(ilogb(n));
221         stack.resize(ilogb(n));
222     #else
223         std::vector<offset> stack;
224         stack.resize(ilogb(n));
225     #endif
226
227     offset s = 0; // stack empty
228     offset j = root<offset>();
229     while (true) {
230         offset k = t[j];
231         sigma[j + 1] = k;
232         if (general_case(j, n)) {
233             offset loser = left_child(j) + (t[j] == t[left_child(j)]);
234             offset winner = left_child(j) + (t[j] != t[left_child(j)]);

```

```

233     stack[s] = winner;
234     ++s;
235     j = loser;
236 }
237 else {
238     offset excess = handle_base_case(t, j, sigma, a, J, less);
239     if (s  $\neq$  0) {
240         --s;
241         j = stack[s];
242         update(t, j, n, excess, a, J, less);
243     }
244     else {
245         sigma[0] = excess;
246         return;
247     }
248 }
249 }
250 }
251
252 template <typename permutation, typename offset, typename iterator,
253           typename index, typename bit_vector>
254 void permute_in_place(permutation& sigma, offset n, iterator a,
255                       index J, bit_vector& b) {
256     using element = typename std::iterator_traits<iterator>::
257         value_type;
258     offset i = 0;
259     while (i < n) {
260         b[i] = false;
261         offset k = sigma[i];
262         if (k == i) {
263             b[i] = true;
264         }
265         ++i;
266     }
267     i = 0;
268     while (i < n) {
269         if (b[i] == false) { // process next cycle
270             b[i] = true;
271             index H = offset_to_index(i, J); // H as in hole
272             element temporary = std::move(*(a + H));
273             offset k = sigma[i];
274             while (k  $\neq$  i) {
275                 b[k] = true;
276                 index K = offset_to_index(k, J);
277                 *(a + H) = std::move(*(a + K));
278                 H = K;
279                 k = sigma[k];
280             }
281             *(a + H) = std::move(temporary);
282         }
283         ++i;
284     }

```

```

282 }
283
284 template <typename iterator, typename comparator>
285 void make_heap(iterator a, iterator past, comparator less) {
286     using index = std::size_t;
287     index n = past - a;
288     if (n < 2) {
289         return;
290     }
291     index lg_n = ilogb(n);
292     index lg_lg_n = ilogb(std::max(index(2), lg_n));
293     index s = 8;
294     index h = 3;
295     while (s < FACTOR * lg_n / lg_lg_n) {
296         s = 2 * s;
297         h = h + 1;
298     }
299     if (n < 2 * s) {
300         make_heap(a, root(), n, less);
301         return;
302     }
303
304 #ifdef IN_PLACE
305     cphstl::packed_array sigma(h + 1);
306     sigma.resize(2 * s);
307     cphstl::packed_array t(h + 1);
308     t.resize(4 * s);
309     cphstl::packed_array b(1);
310     b.resize(2 * s);
311 #else
312     std::vector<int> sigma;
313     sigma.resize(2 * s);
314     std::vector<int> t;
315     t.resize(4 * s);
316     std::vector<unsigned char> b;
317     b.resize(2 * s);
318 #endif
319
320     index const R = (1 << lg_n) - 1;
321     index K = ancestor(R - 1, h - 1);
322     index J = ancestor(n - 1, h - 1);
323     index I = parent(K + 1);
324     if (I < J) {
325         J = parent(J);
326     }
327     while (K > J) {
328         populate_tournament(t, 2 * s - 1);
329         run_tournament(t, 2 * s - 1, a, K, less);
330         convert_tournament(t, 2 * s - 1, sigma, a, K, less);
331         permute_in_place(sigma, s, a, K, b);
332         for (index Z = K; (Z & 1) == 1; ) {
333             Z = parent(Z);

```

```

334     sift_down(a, Z, n, less);
335 }
336 -- K;
337 }
338 make_heap(a, J, n, less);
339 for (index Z = J; (Z bitand 1) == 1; ) {
340     Z = parent(Z);
341     sift_down(a, Z, n, less);
342 }
343 while (K > I) {
344     -- K;
345     populate_tournament(t, 4 * s - 1);
346     run_tournament(t, 4 * s - 1, a, K, less);
347     convert_tournament(t, 4 * s - 1, sigma, a, K, less);
348     permute_in_place(sigma, 2 * s, a, K, b);
349     for (index Z = K; (Z bitand 1) == 1; ) {
350         Z = parent(Z);
351         sift_down(a, Z, n, less);
352     }
353 }
354 }
355 }

```

*tuned\_gm.h++*

```

1 // Gonnet & Munro using extra space for elements
2
3 #include <algorithm> // std::copy std::make_heap
4 #include <cstdint> // std::size_t
5 #include <functional> // std::less
6 #include <iterator> // std::iterator_traits
7 #include <vector> // std::vector
8
9 #ifndef FACTOR
10 #define FACTOR 12
11 #endif
12
13 namespace tuned_gm {
14
15     constexpr std::size_t root() {
16         return 0;
17     }
18
19     template <typename index>
20     index parent(index i) {
21         return (i - 1) / 2;
22     }
23
24     template <typename index, typename height>
25     index ancestor(index i, height h) {
26         return (i + 1) / (1 << h) - 1;
27     }

```

```

28
29 template <typename index>
30 index left_child(index i) {
31     return 2 * i + 1;
32 }
33
34 template <typename index>
35 index right_child(index i) {
36     return 2 * i + 2;
37 }
38
39 template <typename index>
40 index odd(index i) {
41     return (i bitand 1) == 1;
42 }
43
44 template <typename index>
45 index even(index i) {
46     return (i bitand 1) == 0;
47 }
48
49 template <typename index>
50 index sibling(index i) {
51     return i + odd(i) - even(i);
52 }
53
54 template <typename index>
55 bool is_inside(index i, index n) {
56     return i < n;
57 }
58
59 template <typename integer>
60 bool is_power_of_2(integer n) {
61     return __builtin_popcount(n) == 1;
62 }
63
64 template <typename iterator, typename index, typename comparator>
65 void sift_down(iterator a, index i, index n, comparator less) {
66     using element = typename std::iterator_traits<iterator>::
        value_type;
67     element x = std::move(*(a + i));
68     index j = left_child(i);
69     while (is_inside(j, n)) {
70         if (is_inside(j + 1, n)) {
71             j = j + less(*(a + j), *(a + (j + 1)));
72         }
73         if (not less(x, *(a + j))) {
74             break;
75         }
76         *(a + i) = std::move(*(a + j));
77         i = j;
78         j = left_child(i);

```



```

79     }
80     *(a + i) = std::move(x);
81 }
82
83 template <typename iterator, typename index, typename comparator>
84 void make_heap(iterator a, index K, index N, comparator less) {
85     if (not is_inside(left_child(K), N)) {
86         return;
87     }
88     index L = K;
89     while (is_inside(left_child(L), N)) {
90         L = left_child(L);
91     }
92     index R = K;
93     while (is_inside(right_child(R), N)) {
94         R = right_child(R);
95     }
96     if (R < L) {
97         R = right_child(R);
98     }
99     do {
100         L = parent(L);
101         R = parent(R);
102         index J = R;
103         while (true) {
104             sift_down(a, J, N, less);
105             if (J == L) {
106                 break;
107             }
108             J -= 1;
109         }
110     } while (L != R);
111 }
112
113 template <typename input, typename index, typename output>
114 void move_heap(input& a, index j, index n, output& o) {
115     index i = j;
116     index k = j;
117     index p = root();
118     while (i < n) {
119         j = i;
120         while (j ≤ k) {
121             *(o + p) = std::move(*(a + j));
122             ++j;
123             ++p;
124         }
125         i = left_child(i);
126         k = right_child(k);
127     }
128 }
129
130 template <typename tournament, typename index>

```

```

131 void populate_tournament(tournament& t, index n, index J, index E)
132 {
133     index i = parent(n);
134     t[i] = E;
135     ++i;
136     index I = J;
137     index K = J;
138     while (i < n) {
139         J = I;
140         while (J ≤ K) {
141             t[i] = J;
142             ++i;
143             ++J;
144         }
145         I = left_child(I);
146         K = right_child(K);
147     }
148 }
149
150 template <typename tournament, typename index, typename input,
151           typename comparator>
152 void run_tournament(tournament& t, index n, input const& a,
153                    comparator less) {
154     index i = parent(n - 1);
155     while (true) {
156         index C = t[left_child(i)];
157         index D = t[right_child(i)];
158         t[i] = less(*(a + C), *(a + D)) ? D : C;
159         if (i == root()) {
160             break;
161         }
162         -- i;
163     }
164 }
165
166 template <typename tournament, typename index, typename input,
167           typename comparator>
168 void update(tournament& t, index j, index h, index E, input const&
169            a, comparator less) {
170     index M = t[j]; // location of the old champion
171     index k = j;
172     do {
173         k = left_child(k);
174         k = k + (t[k] ≠ M);
175         -- h;
176     } while (h ≠ 0);
177     t[k] = E;
178     do {
179         k = parent(k);
180         index C = t[left_child(k)];
181         index D = t[right_child(k)];
182         t[k] = less(*(a + C), *(a + D)) ? D : C;

```

```

178     } while (k  $\neq$  j);
179 }
180
181 template <typename tournament, typename index, typename input,
182           typename output, typename comparator>
183 index handle_base_case(tournament& t, index j, input& a, output& o,
184                       index J, comparator less) {
185     index champion = t[j];
186     *(o + J) = std::move(*(a + champion));
187     index k = left_child(j);
188     index l = k + (t[k] == champion);
189     index w = sibling(l);
190     index L = left_child(J);
191     index W = right_child(J);
192     index second = t[l];
193     *(o + L) = std::move(*(a + second));
194     k = left_child(l);
195     index ll = k + (t[k] == second);
196     index lw = sibling(ll);
197     *(o + left_child(L)) = std::move(*(a + t[ll]));
198     k = left_child(lw);
199     index lwl = k + (t[k] == second);
200     *(o + right_child(L)) = std::move(*(a + t[lwl]));
201     k = left_child(ll);
202     index lll = k + (t[k] == t[ll]);
203     k = left_child(w);
204     index wl = k + (t[k] == champion);
205     index ww = sibling(wl);
206     k = left_child(ww);
207     index wwl = k + (t[k] == champion);
208     index excess = t[wwl];
209     k = left_child(wl);
210     index wll = k + (t[k] == t[wl]);
211     *(o + left_child(W)) = std::move(*(a + t[wll]));
212     index U = t[wl];
213     index V = t[lll];
214     if (less(*(a + U), *(a + V))) {
215         *(o + W) = std::move(*(a + V));
216         *(o + right_child(W)) = std::move(*(a + U));
217     }
218     else {
219         *(o + W) = std::move(*(a + U));
220         *(o + right_child(W)) = std::move(*(a + V));
221     }
222     return excess;
223 }
224
225 template <typename tournament, typename index, typename input,
226           typename output, typename comparator>
227 index convert_tournament(tournament& t, index j, index h, input& a,
228                          output& o, index P, comparator less) {
229     // P: current location in the output heap

```

```

226 // j: current root in the tournament tree
227 if (h == 3) {
228     return handle_base_case(t, j, a, o, P, less);
229 }
230 index champion = t[j];
231 *(o + P) = std::move(*(a + champion));
232 index k = left_child(j);
233 k = k + (t[k] == champion);
234 index excess = convert_tournament(t, k, h - 1, a, o, left_child(P), less);
235 k = sibling(k);
236 update(t, k, h - 1, excess, a, less);
237 return convert_tournament(t, k, h - 1, a, o, right_child(P), less);
238 }
239
240 template <typename input, typename index, typename tournament,
241           typename output, typename comparator>
242 index make_heap(input& a, index K, index N, index E, tournament& t,
243               output& o, index P, comparator less) {
244     index height = 0;
245     index L = K;
246     while (is_inside(left_child(L), N)) {
247         ++height;
248         L = left_child(L);
249     }
250     index n = 1 << (height + 1);
251     populate_tournament(t, 2 * n - 1, K, E);
252     run_tournament(t, 2 * n - 1, a, less);
253     E = convert_tournament(t, root(), height + 1, a, o, P, less);
254     return E;
255 }
256
257 template <typename iterator, typename comparator>
258 void make_heap(iterator a, iterator past, comparator less) {
259     using index = std::size_t;
260     using element = typename std::iterator_traits<iterator>::
261         value_type;
262     index n = past - a;
263     index lg_n = ilogb(std::max(index(2), n));
264     index s = 8;
265     index h = 3;
266     while (s < FACTOR * lg_n) {
267         s = 2 * s;
268         h = h + 1;
269     }
270     if (n < 2 * s) {
271         make_heap(a, root(), n, less);
272         return;
273     }
274     std::vector<index> t;
275     t.resize(4 * s);

```

```

273     element* o = (element*) malloc(n * sizeof(element)); // no
        element moves
274
275     index const R = (1 << lg_n) - 1;
276     index K = ancestor(R - 1, h - 1);
277     index J = ancestor(n - 1, h - 1);
278     index I = parent(K + 1);
279     if (I < J) {
280         J = parent(J);
281     }
282     index excess = root();
283     if (K == J) {
284     }
285     else if (K == J + 1) {
286         make_heap(a, K, n, less);
287     }
288     else {
289         move_heap(a, K, n, o);
290         -- K;
291         while (K > J) {
292             excess = root();
293             excess = make_heap(a, K, n, excess, t, a, K + 1, less);
294             *(a + root()) = std::move(*(a + excess));
295             for (index Z = K + 1; (Z & 1) == 1; ) {
296                 Z = parent(Z);
297                 sift_down(a, Z, n, less);
298             }
299             -- K;
300         }
301         excess = root();
302         *(o + (s - 1)) = std::move(*(a + excess));
303         excess = make_heap(o, root(), s - 1, s - 1, t, a, K + 1, less);
304         *(a + root()) = std::move(*(o + excess));
305         for (index Z = K + 1; (Z & 1) == 1; ) {
306             Z = parent(Z);
307             sift_down(a, Z, n, less);
308         }
309     }
310     make_heap(a, J, n, less);
311     for (index Z = J; (Z & 1) == 1; ) {
312         Z = parent(Z);
313         sift_down(a, Z, n, less);
314     }
315     if (I == J) {
316     }
317     else if (J == I + 1) {
318         make_heap(a, I, n, less);
319         for (index Z = I; (Z & 1) == 1; ) {
320             Z = parent(Z);
321             sift_down(a, Z, n, less);
322         }
323     }

```

```

324     else {
325         K = J - 1;
326         move_heap(a, K, n, o);
327         -- K;
328         while (K ≥ I) {
329             excess = root();
330             excess = make_heap(a, K, n, excess, t, a, K + 1, less);
331             *(a + root()) = std::move(*(a + excess));
332             for (index Z = K + 1; (Z bitand 1) == 1; ) {
333                 Z = parent(Z);
334                 sift_down(a, Z, n, less);
335             }
336             -- K;
337         }
338         excess = root();
339         *(o + (2 * s - 1)) = std::move(*(a + excess));
340         excess = make_heap(o, root(), 2 * s - 1, 2 * s - 1, t, a, I,
341                             less);
342         *(a + root()) = std::move(*(o + excess));
343         for (index Z = I; (Z bitand 1) == 1; ) {
344             Z = parent(Z);
345             sift_down(a, Z, n, less);
346         }
347         free(o);
348     }
349 }

```

*mr.h++*

```

1 // In-situ/In-place McDiarmid & Reed
2
3 // #define IN_PLACE
4
5 #include <algorithm> // std::copy std::sort
6 #include <cmath> // ilogb
7 #include <cstdint> // std::size_t
8 #include <iterator> // std::iterator_traits
9 #include <utility> // std::move
10 #include <vector> // std::vector
11
12 #ifdef IN_PLACE
13 #include "packed_array.h++" // cphstl::packed_array
14 #endif
15
16 #ifndef FACTOR
17 #define FACTOR 16
18 #endif
19
20 namespace mr {
21
22     constexpr std::size_t root() {

```

```

23     return 0;
24 }
25
26 template <typename index>
27 index parent(index i) {
28     return (i - 1) / 2;
29 }
30
31 template <typename index>
32 index left_child(index i) {
33     return (i << 1) + 1;
34 }
35
36 template <typename index>
37 index right_child(index i) {
38     return (i << 1) + 2;
39 }
40
41 template <typename index, typename height>
42 index ancestor(index i, height h) {
43     return ((i + 1) >> h) - 1;
44 }
45
46 template <typename index>
47 bool is_inside(index i, index n) {
48     return i < n;
49 }
50
51 template <typename iterator, typename index, typename comparator>
52 void sift_up(iterator a, index j, comparator less) {
53     using element = typename std::iterator_traits<iterator>::
54         value_type;
55     element in = std::move(*(a + j));
56     while (j > root()) {
57         index i = parent(j);
58         if (less(*(a + i), in)) {
59             *(a + j) = std::move(*(a + i));
60             j = i;
61         }
62         else {
63             break;
64         }
65     }
66     *(a + j) = std::move(in);
67 }
68
69 template <typename iterator, typename index, typename comparator>
70 void sift_down(iterator a, index i, index n, comparator less) {
71     using element = typename std::iterator_traits<iterator>::
72         value_type;
73     element x = std::move(*(a + i));
74     index j = left_child(i);

```

```

73     while (is_inside(j, n)) {
74         j = j + less(*(a + j), *(a + (j + 1)));
75         if (not less(x, *(a + j))) {
76             break;
77         }
78         *(a + i) = std::move(*(a + j));
79         i = j;
80         j = left_child(i);
81     }
82     *(a + i) = std::move(x);
83 }
84
85 enum {LEFT = 0, RIGHT = 1, UNDEFINED = 2};
86
87 template <typename iterator, typename index, typename comparator,
88           typename bit_vector>
89 void make_heap(iterator a, index K, index N, comparator less,
90               bit_vector& b) {
91     using element = typename std::iterator_traits<iterator>::
92         value_type;
93     index L = K;
94     index R = K;
95     index n = root();
96     while (is_inside(L, N)) {
97         L = left_child(L);
98         n = left_child(n);
99         R = right_child(R);
100     }
101     for (index i = root(); is_inside(i, n); ++i) {
102         b[i] = UNDEFINED;
103     }
104     index m = n;
105     do {
106         L = parent(L);
107         R = parent(R);
108         for (index M = R + 1; M > L;) {
109             -- M;
110             -- m;
111             index J = M;
112             index j = m;
113             index d = 0;
114             while (is_inside(left_child(J), N)) {
115                 ++d;
116                 if (b[j] != UNDEFINED) {
117                     index flag = b[j];
118                     J = left_child(J) + flag;
119                     j = left_child(j) + flag;
120                 }
121             }
122             else {
123                 index flag = less(*(a + left_child(J)), *(a + right_child
124                     (J)));

```



```

121         b[j] = flag;
122         J = left_child(J) + flag;
123         j = left_child(j) + flag;
124     }
125 }
126 while (J > M and less(*(a + J), *(a + M))) {
127     J = parent(J);
128     j = parent(j);
129     -- d;
130 }
131 if (J ≠ M) {
132     element temp = std::move(*(a + M));
133     while (d > 0) {
134         *(a + ancestor(J, d)) = std::move(*(a + ancestor(j, d -
135             1)));
136         index i = ancestor(j, d);
137         b[i] = UNDEFINED;
138         -- d;
139     }
140     *(a + J) = std::move(temp);
141 }
142 } while (L ≠ K);
143 }
144
145 template <typename iterator, typename comparator>
146 void make_heap(iterator first, iterator past, comparator less) {
147     using index = std::size_t;
148     index const n = past - first;
149     if (n < 2) {
150         return;
151     }
152     index const m = (n bitand 1) ? n : n - 1;
153     index lg_m = ilogb(m);
154     index s = 2;
155     index h = 1;
156     while (s < FACTOR * lg_m) {
157         s = 2 * s;
158         h = h + 1;
159     }
160     if (n ≤ s) {
161         std::make_heap(first, past, less);
162         return;
163     }
164
165 #ifdef IN_PLACE
166     cphstl::packed_array extra_space(2);
167     extra_space.resize(2 * s);
168 #else
169     std::vector<unsigned char> extra_space;
170     extra_space.resize(2 * s);
171 #endif

```

```

172
173     index const last_leaf = (1 << lg_m) - 2;
174     index K = ancestor(last_leaf, h - 1);
175     index I = parent(K + 1);
176     while (true) {
177         make_heap(first, K, m, less, extra_space);
178         for (index Z = K; (Z bitand 1) == 1; ) {
179             Z = parent(Z);
180             sift_down(first, Z, m, less);
181         }
182         if (K == I) {
183             break;
184         }
185         -- K;
186     }
187     sift_up(first, n - 1, less);
188 }
189 }

```

## Helper

### *packed\_array.h++*

```

1  /*
2   A packed array stores a sequence a small integers of equal length
3   compactly in memory
4  */
5
6  #include <climits> // CHAR_BIT
7  #include <cstdint> // std::size_t
8  #include <vector> // std::vector
9
10 namespace cphstl {
11
12     class packed_array {
13     public:
14
15         using value_type = std::size_t;
16         using size_type = std::size_t;
17
18         enum {word_size = CHAR_BIT * sizeof(value_type)};
19
20         class reference {
21         private:
22
23             friend class packed_array;
24
25             value_type* word_pointer;
26             value_type offset;
27             value_type mask;
28

```

```

29     reference();
30
31 public:
32
33     reference(packed_array& a, std::size_t position)
34         : word_pointer(&a.data[0] + (position << a.shift) / a.
35           word_size),
36           offset((position << a.shift) bitand (a.word_size - 1)),
37           mask(a.mask) {
38
39     ~reference() {
40
41     // For  $a[i] = x$ ;
42     template <typename integer>
43     reference& operator=(integer x) {
44         value_type y = x;
45         y = y bitand mask;
46         y = y << offset;
47         value_type z = mask << offset;
48         *word_pointer &= ~z;
49         *word_pointer |= y;
50         return *this;
51     }
52
53     // For  $a[i] = a[j]$ ;
54     reference& operator=(reference const& r) {
55         value_type x = *r.word_pointer;
56         x = x >> r.offset;
57         x = x bitand r.mask;
58         value_type y = x bitand mask;
59         y = y << offset;
60         value_type z = mask << offset;
61         *word_pointer &= ~z;
62         *word_pointer |= y;
63         return *this;
64     }
65
66     // For  $x = a[i]$ ;
67     operator value_type() const {
68         value_type x = *word_pointer;
69         x = x >> offset;
70         x = x bitand mask;
71         return x;
72     }
73
74     bool operator==(reference const& r) const {
75         value_type x = (value_type) r;
76         value_type y = (value_type) *this;
77         return x == y;
78     }

```

```

79
80     bool operator!=(reference const& r) const {
81         return not (*this == r);
82     }
83
84     template <typename integer>
85     bool operator==(integer x) const {
86         value_type y = (value_type) *this;
87         return value_type(x) == y;
88     }
89
90     template <typename integer>
91     bool operator!=(integer x) const {
92         return not (*this == x);
93     }
94 };
95
96 friend class reference;
97
98 template <typename integer>
99 explicit packed_array(integer min_width_in_bits = 1) {
100     n = 0;
101     value_type delta = 1;
102     shift = 0;
103     while (delta < min_width_in_bits) {
104         delta = 2 * delta;
105         shift = shift + 1;
106     }
107     mask = (1 << delta) - 1;
108 }
109
110 size_type size() const {
111     return n;
112 }
113
114 template <typename integer>
115 void resize(integer new_size) {
116     n = new_size;
117     data.resize(((n << shift) + word_size - 1) / word_size);
118 }
119
120 template <typename integer>
121 reference const operator[](integer position) const {
122     return reference(*this, position);
123 }
124
125 template <typename integer>
126 reference operator[](integer position) {
127     return reference(*this, position);
128 }
129
130 private:

```

```

131
132     std::vector<value_type> data;
133     size_type n;
134     value_type shift;
135     value_type mask;
136 };
137 }

```

## Drivers

*driver.c++*

```

1  #if not defined(MAXSIZE)
2  #define MAXSIZE 64 * 1024 * 1024 // 715827882
3  #endif
4
5  #include <algorithm> // std::random_shuffle std::make_heap std::sort
6  #include <ctime> // std::clock_t std::clock CLOCKS_PER_SEC
7  #include <functional> // std::less
8  #include <iostream> // std::cout std::cerr
9  #include <iterator> // std::iterator_traits
10 #include <utility> // std::move
11
12 template <typename iterator>
13 bool is_permutation(iterator first, iterator past) {
14     using element = typename std::iterator_traits<iterator>::value_type
15     ;
16     std::sort(first, past);
17     for (iterator q = first; q != past; ++q) {
18         element i = element(q - first);
19         if (*q != i) {
20             std::cerr << i << ": element missing " << *q << " instead" <<
21             std::endl;
22             return false;
23         }
24     }
25     return true;
26 }
27
28 template <typename iterator, typename comparator>
29 bool in_heap_order(iterator first, iterator past, comparator less) {
30     using index = typename std::iterator_traits<iterator>::
31     difference_type;
32     const iterator a = first - 1;
33     const index N = past - first;
34     bool violated = false;
35     for (index i = N; i > 1; i--) {
36         if (less(a[i / 2], a[i])) {
37             std::cerr << i << ": parent=" << a[i / 2] << "; me=" << a[i] <<
38             std::endl;
39             violated = true;
40         }
41     }
42 }

```

```

36     }
37 }
38 return not violated;
39 }
40
41 template <typename iterator, typename index, typename comparator>
42 bool in_heap_order(iterator a, index j, index n, comparator less) {
43     if (j > n) {
44         return true;
45     }
46     bool ok = true;
47     index left = 2 * j;
48     index right = left + 1;
49     if (left ≤ n and less(a[j], a[left])) {
50         std::cerr << a[j] << std::endl;
51         std::cerr << a[left] << " " << a[right] << std::endl;
52         if (2 * left < n) {
53             std::cerr << a[2 * left] << " " << a[2 * left + 1] << " ";
54             std::cerr << a[2 * right] << " " << a[2 * right + 1] << std::
55                 endl;
56         }
57         ok = false;
58     }
59     if (right ≤ n and less(a[j], a[right])) {
60         std::cerr << a[j] << std::endl;
61         std::cerr << a[left] << " " << a[right] << std::endl;
62         if (2 * left < n) {
63             std::cerr << a[2 * left] << " " << a[2 * left + 1] << " ";
64             std::cerr << a[2 * right] << " " << a[2 * right + 1] << std::
65                 endl;
66         }
67         ok = false;
68     }
69     ok &= in_heap_order(a, 2 * j, n, less);
70     ok &= in_heap_order(a, 2 * j + 1, n, less);
71     return ok;
72 }
73
74 #include "algorithm.h++"
75
76 #ifdef MEASURE_COMPARISONS
77 long long volatile comparisons = 0;
78
79 template <typename T>
80 class counting_comparator {
81 public:
82     using first_argument_type = T;
83     using second_argument_type = T;
84     using result_type = bool;
85
86     bool operator()(T const& a, T const& b) const {

```

```

86     ++comparisons;
87     return a < b;
88 }
89 };
90 #endif
91
92 #ifdef MEASURE_MOVES
93 long long volatile moves = 0;
94
95 template <typename T>
96 class move_counter {
97 private:
98
99     T datum;
100
101     move_counter(move_counter const&) = delete;
102     move_counter& operator=(move_counter const&) = delete;
103
104 public:
105
106     explicit move_counter()
107         : datum(0) {
108         moves += 1;
109     }
110
111     template <typename number>
112     explicit move_counter(number x = 0)
113         : datum(x) {
114         moves += 1;
115     }
116
117     move_counter(move_counter&& other) {
118         datum = std::move(other.datum);
119         moves += 1;
120     }
121
122     move_counter& operator=(move_counter&& other) {
123         datum = std::move(other.datum);
124         moves += 1;
125         return *this;
126     }
127
128     operator T() const {
129         return datum;
130     }
131
132     template <typename U>
133     friend bool operator<(move_counter<U> const&, move_counter<U> const
134         &);
135
136     template <typename U>

```

```

136     friend bool operator==(move_counter<U> const& x, move_counter<U>
137                               const& y);
138 };
139
140 template <typename T>
141 bool operator<(move_counter<T> const& x, move_counter<T> const& y) {
142     return x.datum < y.datum;
143 }
144
145 template <typename T>
146 bool operator==(move_counter<T> const& x, move_counter<T> const& y) {
147     return x.datum == y.datum;
148 }
149 #endif
150
151 template <typename iterator>
152 void generate(iterator p, iterator r, char z) {
153     using element = typename std::iterator_traits<iterator>::value_type
154     ;
155     switch (z) {
156     case 'd':
157         for (iterator q = p; q < r; ++q)
158             *q = element((r - 1) - q);
159         break;
160     case 'i':
161         for (iterator q = p; q < r; ++q)
162             *q = element(q - p);
163         break;
164     case 'r':
165         for (iterator q = p; q < r; ++q)
166             *q = element(q - p);
167         std::random_shuffle(p, r);
168         break;
169     case 'z':
170         bool t = false;
171         for (iterator q = p; q < r; ++q) {
172             *q = element(t);
173             t = not t;
174         }
175         break;
176     }
177 }
178
179 void usage(char const* program) {
180     std::cerr << "Usage: " << program
181     << " <N> <'i'ncreasing | 'd'ecreasing | 'r'andom | 'b'ool>"
182     << std::endl;
183     exit(1);
184 }
185
186 int main(int argc, char** argv) {

```



```

186
187 #ifdef MEASURE_MOVES
188     using element = move_counter<int>;
189 #else
190     using element = int;
191 #endif
192
193 #ifdef MEASURE_COMPARISONS
194     using C = counting_comparator<element>;
195 #else
196     using C = std::less<element>;
197 #endif
198
199     unsigned long N = 15;
200     char method = 'i';
201     if (argc == 2) {
202         N = atoi(argv[1]);
203         method = 'i';
204     }
205     else if (argc != 3) {
206         usage(argv[0]);
207     }
208     else {
209         N = atoi(argv[1]);
210         method = *argv[2];
211     }
212     if (N > MAXSIZE) {
213         std::cerr << "N out of bounds [0.."
214             << MAXSIZE
215             << "]"
216             << std::endl;
217         usage(argv[0]);
218     }
219     switch (method) {
220     case 'd':
221     case 'i':
222     case 'r':
223     case 'b':
224         break;
225     default:
226         std::cerr << "Method not in ['d','i','r','b']" << std::endl;
227         usage(argv[0]);
228     }
229
230     element* a = new element[MAXSIZE];
231     element* b = a;
232     for (volatile unsigned long t = MAXSIZE / N; t > 0; t--) {
233         generate(b, b + N, method);
234         b = b + N;
235     }
236
237 #if defined(MEASURE_MOVES)

```

```

238     moves = 0;
239 #elif defined(MEASURE_COMPARISONS)
240     comparisons = 0;
241 #endif
242
243 #if defined(REPETITIONS)
244     unsigned long const repetitions = REPETITIONS;
245 #else
246     unsigned long const repetitions = MAXSIZE / N;
247 #endif
248
249     b = a;
250
251 #if not defined(MEASURE_COMPARISONS) and not defined(MEASURE_MOVES)
252     std::clock_t start = std::clock();
253 #endif
254
255     for (volatile unsigned long t = 0; t < repetitions; ++t) {
256         NAME::make_heap(b, b + N, C());
257         b = b + N;
258     }
259
260 #if not defined(MEASURE_COMPARISONS) and not defined(MEASURE_MOVES)
261     std::clock_t stop = std::clock();
262 #endif
263
264 #if not defined(NDEBUG)
265     b = a;
266     for (volatile unsigned long t = 0; t < repetitions; ++t) {
267         bool ok = in_heap_order(b, b + N, std::less<element>());
268         if (not ok) {
269             return 1;
270         }
271         if (method == 'd' or method == 'i' or method == 'r') {
272             ok = is_permutation(b, b + N);
273             if (not ok) {
274                 return 2;
275             }
276         }
277         b = b + N;
278     }
279 #endif
280
281     double t = double(repetitions) * double(N);
282
283 #if defined(MEASURE_COMPARISONS)
284     std::cout.precision(3);
285     std::cout << N << '\t' << double(comparisons) / t << std::endl;
286 #elif defined(MEASURE_MOVES)
287     std::cout.precision(3);
288     std::cout << N << '\t' << double(moves) / t << std::endl;
289 #else

```

```

290     double ns = 1000000000.0 * double(stop - start) / double(
        CLOCKS_PER_SEC);
291     std::cout.precision(4);
292     std::cout << N << '\t' << ns / t << std::endl;
293 #endif
294
295     delete[] a;
296     return 0;
297 }

```

### *test-driver.c++*

```

1  #include <algorithm> // std::random_shuffle std::make_heap std::sort
2  #include <cassert> // assert macro
3  #include <functional> // std::less
4  #include <iostream> // std::cout std::cerr
5  #include <vector> // std::vector
6
7  template <typename iterator>
8  void show(iterator a, iterator z) {
9      while (a != z) {
10         std::cout << long(*a) << " ";
11         ++a;
12     }
13     std::cout << std::endl;
14 }
15
16 template <typename index>
17 index left_heap_child(index i) {
18     return 2 * i + 1;
19 }
20
21 template <typename index>
22 index right_heap_child(index i) {
23     return 2 * i + 2;
24 }
25
26 template <typename iterator, typename index>
27 void print(iterator a, index j, index N) {
28     if (j < N) {
29         std::cerr << a[j];
30     }
31     std::cerr << " ";
32 }
33
34 template <typename iterator, typename index, typename comparator>
35 bool in_heap_order(iterator a, index j, index n, comparator less) {
36     if (j ≥ n) {
37         return true;
38     }
39     bool ok = true;
40     index left = left_heap_child(j);

```

```

41 index right = right_heap_child(j);
42 if (left < n and less(a[j], a[left])) {
43     std::cerr << "left child not in heap order " << j << std::endl;
44     print(a, j, n);
45     std::cerr << std::endl;
46     print(a, left, n);
47     print(a, right, n);
48     std::cerr << std::endl;
49     print(a, left_heap_child(left), n);
50     print(a, right_heap_child(left), n);
51     print(a, left_heap_child(right), n);
52     print(a, right_heap_child(right), n);
53     std::cerr << std::endl;
54     ok = false;
55 }
56 if (right < n and less(a[j], a[right])) {
57     std::cerr << "right child not in heap order " << j << std::endl;
58     print(a, j, n);
59     std::cerr << std::endl;
60     print(a, left, n);
61     print(a, right, n);
62     std::cerr << std::endl;
63     print(a, left_heap_child(left), n);
64     print(a, right_heap_child(left), n);
65     print(a, left_heap_child(right), n);
66     print(a, right_heap_child(right), n);
67     std::cerr << std::endl;
68     ok = false;
69 }
70 ok &= in_heap_order(a, left_heap_child(j), n, less);
71 ok &= in_heap_order(a, right_heap_child(j), n, less);
72 return ok;
73 }
74
75 template <typename iterator>
76 bool is_permutation(iterator first, iterator past) {
77     using integer = typename std::iterator_traits<iterator>::value_type
78     ;
79     std::vector<integer> v;
80     v.resize(past - first);
81     std::copy(first, past, v.begin());
82     std::sort(v.begin(), v.end());
83     for (auto q = v.begin(); q != v.end(); ++q) {
84         integer i = integer(q - v.begin());
85         if (*q != i) {
86             std::cerr << i << " missing " << *q << " instead" << std::endl;
87             return false;
88         }
89     }
90     return true;
91 }

```

```

92 template <typename iterator>
93 void generate(iterator p, iterator r) {
94     using element = typename std::iterator_traits<iterator>::value_type
95     ;
96     for (iterator q = p; q < r; ++q) {
97         *q = element(q - p);
98     }
99     std::random_shuffle(p, r);
100 }
101 #include "algorithm.h++" // NAME
102
103 int main() {
104     int const magic = 20;
105     int N = (1 << magic);
106     int a[N];
107     std::vector<int> testcase;
108     for (int n = 0; n <= 20000; ++n) { // 8448
109         testcase.push_back(n);
110     }
111     for (int h = 3; h != magic; ++h) {
112         for (int delta = -5; delta < 6; ++delta) {
113             testcase.push_back((1 << h) + delta);
114         }
115     }
116     testcase.push_back(N);
117     for (int n: testcase) {
118         std::cout << "n: " << n << std::endl;
119         generate(a + 0, a + n);
120         NAME::make_heap(a + 0, a + n, std::less<int>());
121         assert(::in_heap_order(a, 0, n, std::less<int>()));
122         assert(::is_permutation(a + 0, a + n));
123     }
124 }

```

## Makefile

### *makefile*

```

1 CXX=g++
2 CXXFLAGS=-O3 -std=c++11 -x c++ -Wall -Wextra -DNDEBUG #-DIN_PLACE
3
4 header-files:= $(wildcard *.h++)
5 versions:= $(basename $(header-files))
6 time-tests:= $(addsuffix .time, $(versions))
7 log-files:= $(addsuffix .log, $(versions))
8 comp-tests:= $(addsuffix .comp, $(versions))
9 move-tests:= $(addsuffix .move, $(versions))
10 branch-tests:= $(addsuffix .branch, $(versions))
11 cache-tests:= $(addsuffix .cache, $(versions))
12 instruction-tests:= $(addsuffix .count, $(versions))

```

```

13 unittests:= $(addsuffix .test, $(versions))
14 portability-tests:= $(addsuffix .port, $(versions))
15 profilings:= $(addsuffix .prof, $(versions))
16
17 N = 1023 32767 1048575 33554431
18 data = r i d b
19
20 $(time-tests): %.time : %.h++
21     @cp *.h++ algorithm.h++
22     $(CXX) $(CXXFLAGS) -DNAME=$* driver.c++
23     @for n in $(N) ; do \
24         ./a.out $$n r ; \
25     done; \
26     rm -f algorithm.h++ ./a.out
27
28 $(log-files): %.log : %.h++
29     @cp *.h++ algorithm.h++
30     $(CXX) $(CXXFLAGS) -DNAME=$* driver.c++
31     @for n in 'cat n.txt' ; do \
32         ./a.out $$n r >> *.log ; \
33     done; \
34     rm -f algorithm.h++ ./a.out
35
36 $(move-tests): %.move : %.h++
37     @cp *.h++ algorithm.h++
38     $(CXX) $(CXXFLAGS) -DMEASURE_MOVES -DNAME=$* driver.c++
39     @for d in $(data) ; do \
40         echo $$d ; \
41         for n in $(N) ; do \
42             ./a.out $$n $$d ; \
43         done \
44     done; \
45     rm -f algorithm.h++ ./a.out
46
47 $(comp-tests): %.comp : %.h++
48     @cp *.h++ algorithm.h++
49     $(CXX) $(CXXFLAGS) -DMEASURE_COMPARISONS -DNAME=$* driver.c++
50     @for d in $(data) ; do \
51         echo $$d ; \
52         for n in $(N) ; do \
53             ./a.out $$n $$d ; \
54         done \
55     done; \
56     rm -f algorithm.h++ ./a.out
57
58 $(instruction-tests): %.count : %.h++
59     @cp *.h++ algorithm.h++
60     @for n in $(N) ; do \
61         python instruction_count.py $* $$n ; \
62         rm -f ./a.out ; \
63         rm -f ./cachegrind.out.* ; \
64     done

```

```

65
66 $(cache-tests): %.cache : %.h++
67   @for n in $(N) ; do \
68     python cache_misses.py driver.c++ $* $$n ; \
69     rm -f ./a.out ; \
70     rm -f ./cachegrind.out.* ; \
71   done
72
73 $(branch-tests): %.branch : %.h++
74   @for n in $(N) ; do \
75     python branch_mispredictions.py $* $$n ; \
76     rm -f ./a.out ; \
77     rm -f ./cachegrind.out.* ; \
78   done
79
80 TESTFLAGS=-O3 -std=c++11 -Wall -Wextra -x c++ -g -DDEBUG
81
82 $(unittests): %.test : %.h++
83   @cp $*.h++ algorithm.h++
84   $(CXX) $(TESTFLAGS) -DNAME=$* test-driver.c++
85   ./a.out
86   rm -f algorithm.h++ ./a.out
87
88 $(portability-tests): %.port : %.h++
89   @cp $*.h++ algorithm.h++
90   $(CXX) $(TESTFLAGS) -DNAME=$* portability-driver.c++
91   ./a.out
92   rm -f algorithm.h++ ./a.out
93
94 PROFILERFLAGS = -DNDEBUG -Wall -std=c++11 -pedantic -x c++ -g
95
96 $(profilings): %.prof : %.h++
97   @cp $*.h++ algorithm.h++
98   $(CXX) $(PROFILERFLAGS) -DNAME=$* -DMAXSIZE=4194300 driver.c++
99   valgrind --tool=callgrind --dump-instr=yes --collect-jumps=yes
100     --callgrind-out-file=$*.callgrind.out ./a.out 1048575
101   rm -f algorithm.h++
102
103 # Other tools
104
105 clean:
106   - rm -f a.out temp algorithm.h++ 2>/dev/null
107
108 veryclean: clean
109   - rm -f *~ */*~ 2>/dev/null
110
111 find:
112   find . -type f -print -exec grep $(word) {} \; | less

```